Universidade Federal de Minas Gerais
DCC023: Redes de Computadores
Trabalho Prático
PeeringDB REST API

Jonatas Cavalcante - 2014004301

1. Execução

O trabalho é composto por dois arquivos: *server.py* e *client.py*. Além destes, há uma pasta com os testes disponibilizados para o desenvolvimento do trabalho (*tp3-input*). Primeiramente, é necessário iniciar o servidor para testar a aplicação. Para isso, utilize o seguinte comando:

```
python server.py port Netfile Ixfile Netixlanfile
```

Onde [port] é o porto no qual o servidor receberá mensagens, [Netfile] é o caminho do arquivo de redes, [Ixfile] é o caminho do arquivo de IXPs e [Netixlanfile] é o caminho do arquivo de associações.

Uma vez que o servidor está inicializado, para fazer os testes da aplicação, utilize a seguinte linha de comando:

```
python client.py IP:port Opt
```

Onde [IP] é o IP do servidor, [port] é o porto que o servidor está utilizando e [Opt] especifica o número da análise: "0" se for a análise de IXPs por rede ou "1" se for a análise redes por IXP.

Para finalizar a execução do servidor, basta apenas entrar com o comando CTRL+C no terminal. O programa cliente, ao terminar de realizar a análise escolhida na linha de terminal, encerra sua execução automaticamente.

2. Funcionamento

2.1. Servidor

A primeira tarefa do servidor é ler os parâmetros passados na linha de comando em sua inicialização. Uma vez que os parâmetros são lidos, o servidor carrega os dados JSON de cada um dos três arquivos e os armazena em variáveis do programa.

Após isso, o servidor estabelece a definição de cada uma das três *endpoints* solicitadas para esse trabalho prático. São elas:

`/api/ix`: todos os IXPs
- Requisição ao endpoint `/api/ix`
- Resposta: `{"data": <lista dos objetos IXPs>}`

`/api/ixnets/{ix_id}`: identificadores das redes de um IXP
- Requisição ao endpoint `/api/ixnets/{ix_id}`
- Resposta: `{"data": <lista dos identificadores das redes do IXP identificado por 'ix_id'>}`

`/api/netname/{net_id}`: nome de uma rede

- Requisição ao endpoint `/api/netname/{net_id}`
- Resposta: `{"data": <nome da rede identificada por 'net_id'>}`

Com o uso do web framework Flask, a implementação das endpoints foram realizadas, conforme o trecho de código abaixo demonstra. A primeira endpoint consiste em apenas retornar todos os objetos IXP contidos no primeiro arquivo passado como parâmetro na linha de comando. A segunda endpoint consiste em percorrer todos os objetos do terceiro arquivo passado como parâmetro e adicionar o *net_id* da rede caso o campo *ix_id* dela seja igual ao *ix_id* passado como parâmetro na chamada da endpoint. A última endpoint consiste e percorrer o segundo arquivo passado na lista de parâmetros no terminal e, uma vez encontrada a rede em que seu *id* seja igual ao *net_id* passado para a endpoint, retorna o campo *name* da rede.

```
@app.route("/api/ix")
def get_all_IXPs_objects():
    return jsonify(data = ixfile_data['data'])


@app.route("/api/ixnets/<ix_id>")
def get_IXP_networks_ids(ix_id):
    ids = []

    for network in netixlanfile_data['data']:
        if network['ix_id'] == int(ix_id):
            ids.append(network['net_id'])

    return jsonify(data = ids)


@app.route("/api/netname/<net_id>")
def get_network_name(net_id):
    for network in netfile_data['data']:
        if network['id'] == int(net_id):
            return jsonify(data = network['name'])
```

Figura 1 - Trecho de código da implementação das endpoints

Para retornar os dados em JSON, foi utilizado o jsonify do Flask (`from flask import jsonify`).

2.2. Cliente

A implementação do cliente consiste de três funções, a saber:

- `make_request(endpoint)`

Essa função é responsável por estabelecer a conexão entre o cliente e servidor para realizar as requisições necessárias. Primeiramente, o cliente lê os parâmetros passados na linha de comando quando é executado. Após ler esses dados, ele configura dos dados do socket e realiza a conexão com o servidor.

Uma vez estabelecida a conexão, é montada a requisição desejada. O valor da endpoint passada por parâmetro para essa função é usada para montar a requisição, juntamente com IP do servidor e a porta. Uma vez montada a requisição, ela é realizada e a resposta é salva na variável *response*.

Terminada a execução da requisição, fecha-se a conexão e o valor recebido do servidor é tratado, uma vez que a resposta possui o corpo dela e o cabeçalho. Uma vez separados, ela retorna o corpo da resposta, como mostrado no código abaixo:

```
def make_request(endpoint):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = (SERVER_IP, int(PORT))
    client_socket.connect(server_address)

    request = 'GET ' + endpoint + ' HTTP/1.1\r\nHost: ' + SERVER_IP + ':' + PORT + '\r\nContent-Type: application/json\r\n\r\n'
    client_socket.send(request)
    response = ''
    while True:
        recv = client_socket.recv(1024)
        if not recv:
            break
        response += recv

    client_socket.close()
    return response.split("\n\r")[1]
```

Figura 2 - Implementação da função make_request(endpoint)

- `ixps_by_network()`

Essa função é responsável por realizar a primeira análise pedida no trabalho prático (IXPs por redes). Para implementar esse cálculo, o cliente obtém todos os IXPs (através da primeira endpoint) e, percorrendo cada um dos objetos retornados, com o auxílio de um dicionário, realiza a contagem da quantidade de redes associadas àquele IXP. Isso é feito realizando a requisição da segunda endpoint disponibilizada pelo servidor. Para cada rede retornada, verifica-se se ela já está no dicionário. Em caso afirmativo, o valor daquele id de rede é incrementado e, em caso negativo, ele é atribuído valendo 1. Após isso, tem-se um dicionário com todas as redes e com sua respectiva quantidade de IXPs, com o formato par-valor (net_id : qtd_ixp). Para exibir o nome da rede, é feita uma requisição da terceira endpoint do servidor, passando o *net_id* como parâmetro e exibindo a resposta na tela, juntamente com os demais dados, como mostrado no trecho abaixo:

```
def ixps_by_network():
    all_nets = {}
    ixps_data = make_request('/api/ix')
    ixps = json.loads(ixps_data)

    for ixp in ixps['data']:
        nets_data = make_request('/api/ixnets/' + str(ixp['id']))
        nets = json.loads(nets_data)
        for net in nets['data']:
            if net in all_nets:
                all_nets[net] += 1
            else:
                all_nets[net] = 1

    for key, qtd in all_nets.iteritems():
        netname_data = make_request('/api/netname/' + str(key))
        netname = json.loads(netname_data)
        print str(key) + '\t' + netname['data'] + '\t' + str(qtd) + '\n'
```

Figura 3 - Implementação da função ixps_by_network()

- `networks_by_ixp()`

Essa função é responsável por realizar a segunda análise pedida no trabalho prático (redes por IXP). Para implementar esse cálculo, o cliente obtém todos os IXPs (através da primeira endpoint) e, percorrendo cada um dos objetos retornados, realiza a requisição da segunda endpoint do servidor, obtendo um *array* com todas as redes associadas ao IXP corrente. A quantidade de redes associadas ao IXP é simplesmente o tamanho desse vetor retornado. Assim sendo, com o resultado dessa requisição para cada IXP, é exibido na tela esses dados, como mostrado abaixo:

```
def networks_by_ixp():
    ixps_data = make_request('/api/ix')
    ixps = json.loads(ixps_data)

    for ixp in ixps['data']:
        nets_data = make_request('/api/ixnets/' + str(ixp['id']))
        nets = json.loads(nets_data)
        print str(ixp['id']) + '\t' + ixp['name'] + str(len(nets['data'])) + '\n'
```

Figura 4 - Implementação da função networks_by_ixp()

Finalizando o programa cliente, temos apenas uma condicional que verifica o parâmetro Opt passado na linha de comando. Caso seja 0, é chamada a função `ixps_by_network()` e, caso seja 1, a função `networks_by_ixp()` é chamada.

```
if(int(OPT) == 0):
    ixps_by_network()
elif(int(OPT) == 1):
    networks_by_ixp()
```

Figura 5 - Trecho final de código do *client.py*